Title	Improving the Competitive Ratio of the Online OVSF Code Assignment Problem
Author(s)	Miyazaki, Shuichi; Okamoto, Kazuya
Citation	Algorithms (2009), 2(3): 953-972
Issue Date	2009-07-17
URL	http://hdl.handle.net/2433/226991
Right	© 2009 by the authors; licensee Molecular Diversity Preservation International, Basel, Switzerland.; This is an open access article distributed under the Creative Commons Attribution License (CC BY 3.0).
Type	Journal Article
Textversion	publisher

Algorithms **2009**, *2*, 953-972; doi:10.3390/a2030953

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Improving the Competitive Ratio of the Online OVSF Code Assignment Problem*

Shuichi Miyazaki [1] **and Kazuya Okamoto** [2,⋆]

[1] Academic Center for Computing and Media Studies, Kyoto University, Yoshida Honmachi, Sakyo-ku, Kyoto 606-8501, Japan; E-mails: shuichi@media.kyoto-u.ac.jp

[2] Graduate School of Informatics, Kyoto University, Yoshida Honmachi, Sakyo-ku, Kyoto 606-8501, Japan

⋆ Author to whom correspondence should be addressed; E-mail: okia@kuis.kyoto-u.ac.jp

Received: 7 May 2009; in revised form: 14 July 2009 / Accepted: 14 July 2009 / Published: 17 July 2009

Abstract: Online OVSF code assignment has an important application to wireless communications. Recently, this problem was formally modeled as an online problem, and performances of online algorithms have been analyzed by the competitive analysis. The previous best upper and lower bounds on the competitive ratio were 10 and 5/3, respectively. In this paper, we improve them to 7 and 2, respectively. We also show that our analysis for the upper bound is tight by giving an input sequence for which the competitive ratio of our algorithm is $7 - \varepsilon$ for an arbitrary constant $\varepsilon > 0$.

Keywords: online OVSF code assignment; online algorithm; competitive analysis

1. Introduction

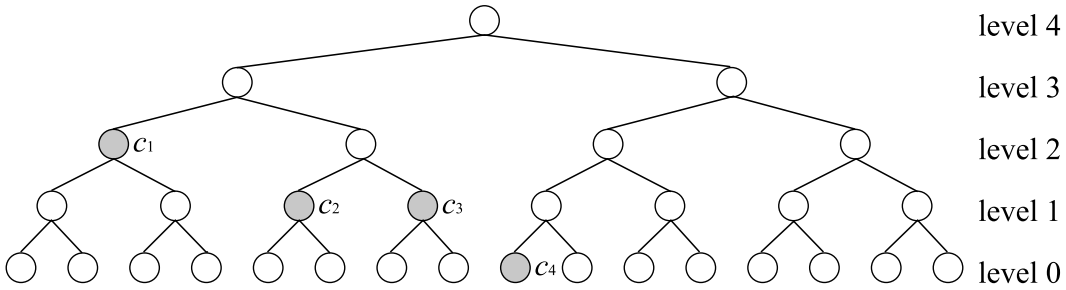
Universal Mobile Telecommunication System (UMTS) [1, 2] is one of the third generation (3G) technologies, which is a mobile communication standard. UMTS uses a high-speed transmission protocol Wideband Code Division Multiple Access (W-CDMA) as the primary mobile air interface. W-CDMA is implemented based on Direct Sequence CDMA (DS-CDMA), which allows several users to commu-

*A preliminary version of this paper was presented at the 19th International Symposium on Algorithms and Computation, ISAAC 2008. This work was supported by KAKENHI 17700015, 19200001, and 20300028.

nicate simultaneously over a single communication channel. DS-CDMA utilizes Orthogonal Variable Spreading Factor (OVSF) code to separate communications [3].

OVSF code is based on an *OVSF code tree*, which is a complete binary tree of height h. The leaves of the OVSF code tree are of level 0 and parents of vertices of level ℓ ($\ell = 0, \ldots, h-1$) are of level $\ell + 1$. Therefore the level of the root is h. Figure 1 shows an OVSF code tree of height 4. (Ignore "$c_1, \ldots, c_4$" and shaded vertices for a while. We use them later.)

Figure 1. An OVSF code tree of hight 4.



Each vertex of level ℓ corresponds to a code of level ℓ. In DS-CDMA, each communication uses a code of the specific level. To avoid interference, we need to assign codes to vertices of an OVSF code tree so that they are *mutually orthogonal*, namely, in any path from the root to a leaf of an OVSF code tree, there is at most one assigned vertex. For example, consider the OVSF code tree given in Figure 1. Let $\ell(c)$ denote the level specified by a code c. Suppose that four requests for inserting codes c_1, c_2, c_3, and c_4 such that $\ell(c_1) = 2$, $\ell(c_2) = \ell(c_3) = 1$, $\ell(c_4) = 0$ arrive in this order, and that they are assigned as depicted in Figure 1. Then, a request for deleting c_2 arrives and c_2 is removed. Next, a request for inserting c_5 such that $\ell(c_5) = 3$ arrives, but it cannot be assigned unless other codes are reassigned. If c_4 is reassigned to one of the children of the vertex to which c_2 was assigned, we can assign c_5 to the right vertex of level 3. In this case, we need 6 assignments (5 assignments and 1 reassignment). However, if c_2 has been assigned to a vertex in the right subtree of the root, and c_4 has been assigned to a vertex in the left subtree, then we need only 5 assignments.

Erlebach *et al.* [4] first modeled this problem as an online problem of minimizing the number of assignments, called the online OVSF code assignment problem, and verified the efficiency of online algorithms using competitive analysis. The *competitive ratio* of an online algorithm ALG is defined as $\max_\sigma\{\frac{E_{ALG}(\sigma)}{E_{OPT}(\sigma)}\}$, where $E_{ALG}(\sigma)$ and $E_{OPT}(\sigma)$ are the costs of ALG and an optimal offline algorithm, respectively, for a sequence σ of requests, and $\max$ is taken over all σ. For the online OVSF code assignment problem, the number of assignments is regarded as a cost. If the competitive ratio of ALG is at most α, we say that ALG is α-competitive. Erlebach *et al.* [4] developed a $\Theta(h)$-competitive algorithm (recall that h is the height of an OVSF code tree), and proved that the lower bound on the competitive ratio of the problem is 1.5. Forišek *et al.* [5] developed a $\Theta(1)$-competitive algorithm, but they did not obtain a concrete constant. Chin, Ting, and Zhang [6] proposed algorithm LAZY by modifying the algorithm of Erlebach *et al.* [4], and proved that the competitive ratio of LAZY is at most 10. Chin, Ting, and Zhang [6] also showed that no online algorithm can be better than 5/3-competitive.

This problem also has an application in assigning subnets to users in computer network managements. An IP address space can be divided into subnets, each of which is a fragment of the whole IP address space consisting of a set of continuous IP addresses of size power of 2. This structure can be represented as a complete binary tree, in exactly the same way as our problem. Usually, the sizes of subnets requested by users depend on the number of computers they want to connect to the subnet, and the task of system managers is to assign subnets to users so that no two assigned subnets overlap. Apparently, we want to minimize the number of reassignments because a reassignment causes a large cost for updating configurations of computers.

Our Contribution. In this paper, we improve both upper and lower bounds on the competitive ratio of this problem, namely, we give a 7-competitive algorithm EXTENDED-LAZY, and show that no online algorithm can be better than 2-competitive. We further show that our upper bound analysis is tight by giving a sequence of requests for which the competitive ratio of EXTENDED-LAZY is $7 - \varepsilon$ for an arbitrary constant $\varepsilon > 0$.

We briefly explain an idea of improving the upper bound. Erlebach *et al.* [4] defined the "compactness" of assignments, and their algorithm keeps an assignment compact at any time. They proved that serving a request, namely assigning (or removing) a code and modifying the assignment to keep it compact, will cause at most one reassignment at each level, which leads to $\Theta(h)$-competitiveness. Chin, Ting, and Zhang [6] pointed out that always keeping assignments compact is too costly. Their algorithm LAZY does not always keep compactness but makes assignments compact when it is necessary. To achieve this relaxation, they defined a "tank", which is a vertex that *virtually* contains a code of a lower level. By exploiting the idea of tanks, they proved that the cost of serving each request is at most 5, which results in 10-competitiveness. Our algorithm, called EXTENDED-LAZY, follows this line. We further relax the compactness by defining "semi-compactness". We also use amortized cost analysis, and prove that serving one insertion (deletion, respectively) and keeping the semi-compactness costs at most 4 (3, respectively). This gives a 7-competitiveness of EXTENDED-LAZY.

Related Results. For the online OVSF code assignment problem, there have been some resource augmentation models, namely, online algorithms are allowed to use more bandwidth than an optimal offline algorithm: Erlebach *et al.* [4] developed a 4-competitive algorithm in which an online algorithm can use a double-sized OVSF code tree. Chin, Zhang, and Zhu [7] developed a 5-competitive algorithm that uses 1/8 extra bandwidth. Recently, Chan *et al.* [8] gave a 1-competitive online algorithm which uses $(h+1)/2$ trees and showed that there is no 1-competitive online algorithm that uses less than $(h+1)/2$ trees. They also gave a 2-competitive algorithm with $3h/8 + 2$ trees and an amortized $(4/3 + \alpha)$-competitive algorithm with $(11/4 + 4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$.

In addition to theoretical analysis, Karakoc and Kavak [9] evaluated the performance of genetic algorithm and simulated annealing by experiments.

Also, there have been several *offline* results. Minn and Siu [10] developed a greedy algorithm for the problem of finding the minimum number of reassignments to modify the current assignment so that the new code can be assigned, given a current assignment configuration of the tree and a new insertion request. Moreover, Erlebach *et al.* [4] proved that this problem is NP-hard and developed a $\Theta(h)$-approximation algorithm. Erlebach, Jacob, and Tomamichel [11] proved that the problem of finding a

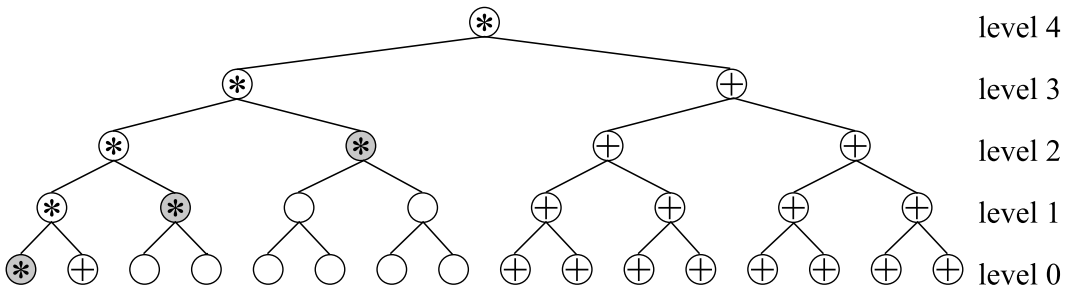
sequence of operations that minimizes the number of reassignments, given an OVSF code tree and a whole sequence of requests, is NP-hard and gave an exponential-time algorithm.

2. Preliminaries

An instance of the online OVSF problem consists of an OVSF code tree of height h and a sequence σ of requests. Each request is either an *insertion* or a *deletion*. An insertion i_c specifies a code c. Upon receiving an insertion i_c, the task of an online algorithm is to assign c to one of the vertices of level $\ell(c)$ (i.e. the level specified by c) of the OVSF code tree, so that the orthogonality condition is not broken. An online algorithm may also reassign other codes (already existing in the tree). A deletion d_c specifies a code c which was previously assigned and has still been assigned to the current OVSF code tree. When a deletion d_c arrives, the task of an online algorithm is to merely remove c from the tree (and similarly, it may reassign other codes in the tree). Each assignment and reassignment causes a cost of one, but removing a code causes no cost. Without loss of generality, we may assume that σ does not include an insertion that cannot be assigned by any reassignment of the existing codes (in other words, the total bandwidth of codes which should be assigned at any point never exceeds the capacity).

We define terminologies needed to give our algorithm, most of which are taken from [6]. Given an assignment configuration, we say that vertex v is *assignable* if none of descendants, ancestors, or v itself is assigned; in other words, a code can be assigned to v without breaking the orthogonality. We also say that vertex v is *dead* if v or one of its descendants is assigned. In the example of Figure 2, shaded vertices are assigned, vertices with pluses ($+$) are assignable, and vertices with stars ($*$) are dead. Note that assignable vertices are all non-dead. If all the assigned vertices are mutually orthogonal and, at any level, all the left vertices (on the same level) to the rightmost dead vertex are dead, then the assignment is called *compact*. For example, the assignment in Figure 2 is compact.

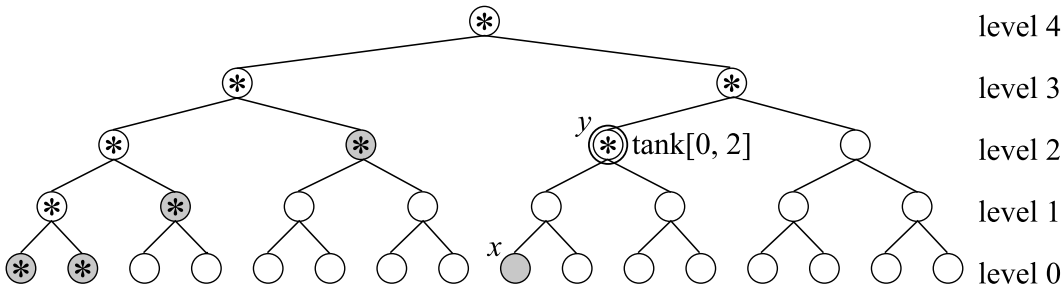
Figure 2. A compact assignment.



Next, we define statuses of levels. Level ℓ is said to be *rich* if a code of level ℓ can be assigned to the leftmost non-dead vertex v at ℓ without reassigning other codes; in other words, v is assignable. Otherwise, the level ℓ is said to be *poor*. For example, in the assignment of Figure 2, levels 0, 2, and 3 are rich and levels 1 and 4 are poor. A level ℓ is said to be *locally rich* if the rightmost assigned vertex is the left child of its parent. For example, in Figure 2, only level 0 is locally rich. Note that in a compact assignment, locally rich levels are always rich.

We introduce a tank, which is taken from [6]: Suppose that two codes c_1 and c_2 of level 0 arrive when the current configuration is in Figure 2. We can assign c_1 to the leftmost assignable vertex of level 0 with keeping the compactness. However, we cannot assign c_2 with keeping the compactness unless we reassign other codes. If c_2 were a code of level 2, it could be assigned to the leftmost assignable vertex of level 2 without breaking the compactness. The idea of a tank is to do this virtually. See Figure 3 for example. The code c_2 is assigned to x, but we consider as if it were assigned to y, and by doing so, we regard the assignment compact. Formally, we virtually assign a code of level b to a vertex v of higher level t ($> b$). In this case, v is called a tank and denoted tank$[b, t]$. Levels b and t are called the *bottom* and the *top* of tank$[b, t]$, respectively. We say that level ℓ ($b \leq \ell \leq t$) *belongs to* tank$[b, t]$. Note that in the definitions and proofs below, we regard y as "assigned" and x as "unassigned".

Figure 3. A semi-compact assignment.



We also define the *semi-compactness*. An assignment is said to be *semi-compact* if the following five conditions are satisfied: (i) All the assigned vertices are mutually orthogonal; (ii) All the left vertices of the rightmost dead vertex are dead at each level; (iii) Each level belongs to at most one tank; (iv) Suppose that there is a tank v(=tank$[b, t]$) at level t. Then level t contains at least one assigned vertex other than v, and there is no dead vertex to the right of v at t; (v) Levels belonging to tanks are all poor except for the top levels of tanks. Figure 3 shows an example of a semi-compact assignment.

3. Algorithm EXTENDED-LAZY

To give a complete description of EXTENDED-LAZY, we first define the following four functions [6]. Note that a single application of each function keeps the orthogonality, but may break the semi-compactness. However, EXTENDED-LAZY combines functions so that the combination keeps the semi-compactness.

- AppendRich(ℓ, c): This function is available if the level of code c is less than or equal to ℓ (namely $\ell(c) \leq \ell$), and level ℓ is rich. It assigns c to the leftmost non-dead vertex at ℓ. Note that if $\ell(c) \neq \ell$, this function creates tank[$\ell(c)$, ℓ].

- AppendPoor(ℓ, c): This function is available if $\ell(c) \leq \ell$ and level ℓ is poor. It assigns code c to the leftmost non-dead vertex v at ℓ. If there is no such v, abort. Note that if $\ell(c) \neq \ell$, tank[$\ell(c)$, ℓ] is created. Then, it releases a code assigned to a vertex in the path from v to the root and returns it.

(Such a code exists because ℓ was poor and v was non-dead. This code is unique because of the orthogonality.)

- FreeTail(ℓ): Release the code assigned to the rightmost assigned vertex at level ℓ, and return it.
- AppendLeft(ℓ, c): This function is available if $\ell(c) = \ell$. Assign code c to the leftmost assignable vertex at level ℓ.

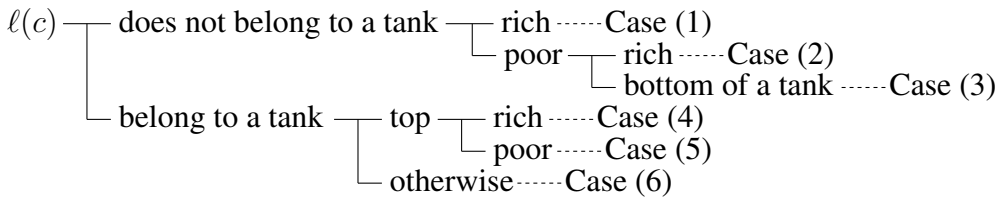
Each of AppendRich, AppendPoor, and AppendLeft yields a cost of 1, and FreeTail does not yield a cost.

Now, we are ready to describe EXTENDED-LAZY. Its behaviors on insertions and deletions are given in Sects. 3.1. and 3.2., respectively. Executions of EXTENDED-LAZY is divided into several cases. In the description of each case, we explain the behavior of EXTENDED-LAZY, and in addition, for the later analysis, we will calculate the cost incurred and an upper bound on the increase in the number of locally rich levels due to the operations.

3.1. Executions of EXTENDED-LAZY *for insertions*

As summarized in Figure 4, the behavior of EXTENDED-LAZY for an insertion i_c is divided into six cases based on the status of the level $\ell(c)$.

Figure 4. Execution of EXTENDED-LAZY for an insertion i_c.



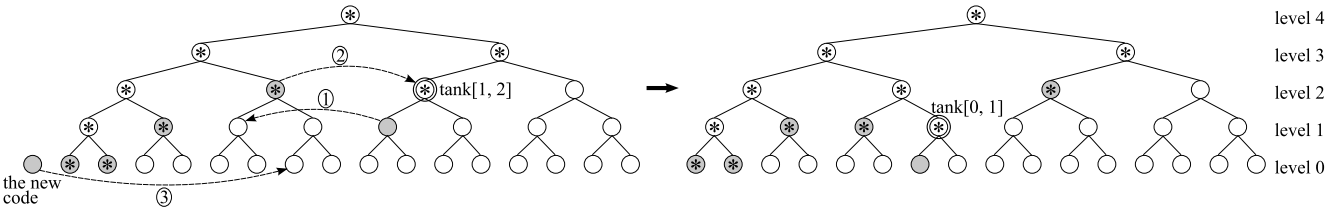
Case (1): The case that $\ell(c)$ does not belong to a tank and is rich. Execute AppendRich($\ell(c), c$). The execution of this case costs 1 and the number of locally rich levels increases by at most one because only $\ell(c)$ changes its status.

Case (2): The case that $\ell(c)$ does not belong to a tank and is poor. Furthermore, if we look at the higher levels than $\ell(c)$ in the order of $\ell(c)+1, \ell(c)+2, \ldots, h$ until we encounter a level that is rich or a bottom of a tank, we encounter a rich level (say, the level t) before a bottom of a tank. In this case, execute AppendRich(t, c). Note that the new tank[$\ell(c), t$] is created. This case costs 1 and the number of locally rich levels increases by at most one since only level t changes its status.

Case (3): The same as Case (2), but when looking at higher levels, we encounter a bottom b of tank[b, t] before we encounter a rich level. Since this case is a little bit complicated, we give an example in Figure 5, in which $\ell(c) = 0$, $b = 1$, and $t = 2$. First, execute FreeTail(t) and receive the code c' of level b (because a level-b code was assigned to level t by exploiting a tank), and then execute AppendPoor(b, c') (note that b is poor by the condition (v) of semi-compactness)(Figure 5①). Next, receive another code c'' of level s. Note that there is an assigned vertex at t because of the condition (iv), and hence $b < s \leq t$.

Then, using AppendRich(t, c''), assign c'' to the vertex which was tank$[b, t]$, which creates the new tank$[s, t]$ if $s \neq t$ (Figure 5②). Now, recall that the level b was poor, and hence c' was assigned to a left child. So, the level b is currently locally rich. We execute AppendRich(b, c) (Figure 5③). Note that tank$[\ell(c), b]$ is newly created. In this case, the cost incurred is 3, and the number of locally rich levels does not change.

Figure 5. An execution of Case (3).



Case (4): The case that $\ell(c)$ is the top of a tank and is rich. First, execute FreeTail($\ell(c)$) and receive the code c' from the bottom of the tank. Then, execute AppendRich($\ell(c), c$) and AppendRich($\ell(c), c'$) in this order. Intuitively speaking, we shift the top of the tank to the right, and assign c to the vertex which was the tank. In this case, it costs 2 and similarly as Case (1), the number of locally rich levels increases by at most one.

Case (5): The case that $\ell(c)$ is the top of a tank, say tank$[b, \ell(c)]$, and is poor. Execute FreeTail($\ell(c)$) and receive the code c' of level b from tank$[b, \ell(c)]$, and execute AppendRich($\ell(c), c$) to process c. Note that $\ell(c')$ $(= b)$ is poor because $\ell(c')$ was the bottom of tank$[b, \ell(c)]$. Also, note that b currently does not belong to a tank. Hence our current task is to process c' of level b where b does not belong to a tank and is poor. So, according to the statuses of levels higher than b, we execute one of Cases (2) or (3). Before going to Cases (2) or (3), the cost incurred is 1 and there is no change in the number of locally rich levels. Hence, the total cost of the whole execution of this case can be obtained by adding one to the cost incurred by the subsequently executed case (Cases (2) or (3)), and the change in the locally rich levels is the same as that of the subsequently executed case.

Case (6): The case that $\ell(c)$ belongs to tank$[b, t]$ and is not the top of tank$[b, t]$. Execute FreeTail(t) and receive the code c' of level b from tank$[b, t]$. Then, execute AppendPoor($\ell(c), c$) and receive a code c'' of level s. By a similar observation as Case (3), we can see that $\ell(c) < s \leq t$. Then, assign c'' to the vertex which was tank$[b, t]$ using AppendRich(t, c''). (Note that tank$[s, t]$ is created if $s \neq t$.) Since $\ell(c)$ was poor, c was assigned to a left child. Hence $\ell(c)$ is currently locally rich. Execute AppendRich($\ell(c), c'$), which creates tank$[b, \ell(c)]$ if $\ell(c) \neq b$. The execution of this case costs 3, and the number of locally rich levels does not change.

Here we give one remark. Suppose that $\ell(c)$ does not belong to a tank and is poor. Then, we look at higher levels to see which of Cases (2) and (3) is executed. But, it may happen that we encounter neither a rich level nor a bottom of a tank, namely there is no tank in upper levels and all the upper levels are poor. We define this case as Case (7). However, we can prove the following lemma.

Lemma 1. *If the semi-compactness is preserved and Case (7) occurs, then there is no configuration that assigns all the current codes.*

Proof. We define the *bandwidth of a cade* c as $2^{\ell(c)}$ and the *bandwidth of the OVSF code tree* of hight h as 2^h. Suppose that the semi-compactness is preserved and Case (7) occurs when an insertion i_c arrives. We show that $2^{\ell(c)}$, the bandwidth required by this request, exceeds the bandwidth of the tree not used by the currently assigned codes.

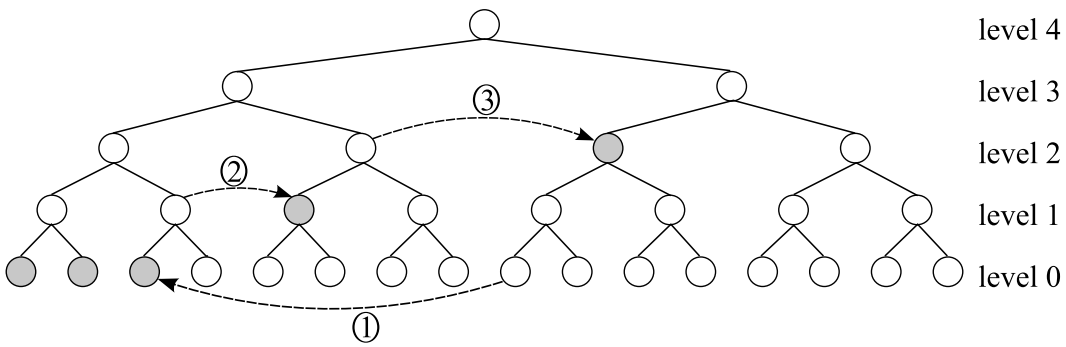
First, we transform the semi-compact assignment to a compact assignment. We remove tanks from the semi-compact assignment. Consider tank$[b,t]$. Release the code $\bar{c}$ of level b from tank$[b,t]$ temporarily. Then, assign $\bar{c}$ to the vertex v immediately right of the rightmost dead vertex at b. Because of conditions (ii) and (v) of semi-compactness, b is poor and the path from v to the root includes exactly one assigned vertex v' at some level ℓ ($\ell \leq t$). Release code $\bar{c}'$ assigned to v' and assign $\bar{c}'$ to the vertex v'' immediately right of the rightmost dead vertex at ℓ. By repeating the above operations, a code $\bar{c}''$ assigned to a vertex at level t is released. ($\bar{c}''$ exists because of condition (iv) of semi-compactness.) We assign $\bar{c}''$ to the vertex which was tank$[b,t]$.

We show that conditions of semi-compactness remain satisfied after the above operation. Conditions (iii), (iv), and (v) remain satisfied for the levels which did not belong to tank$[b,t]$ because no code was released from nor assigned to these levels. Also, conditions (iii), (iv), and (v) remain satisfied for levels which belonged to tank$[b,t]$ because these conditions state on levels that belong to tanks but these levels do not belong to a tank any more since tank$[b,t]$ was removed. (This argument is used several times in the later proofs.)

Condition (i) remains satisfied because once two codes overlap, the one, say c, at the higher level is moved immediately. Suppose that c is moved from v_1 to v_2 in this step. Then, v_1 becomes unassigned but is still dead because one of its descendants is assigned. Also, v_2 was the leftmost non-dead vertex at level $\ell(c)$. Hence, Condition (ii) remains satisfied.

If we do the above operation for each tank independently, the initial semi-compact assignment is transformed to a semi-compact assignment with no tanks. By definition, a semi-compact assignment with no tanks is a compact assignment. For example, the semi-compact assignment in Figure 3 is transformed to the compact assignment in Figure 6.

Figure 6. Transformation from a semi-compact assignment to a compact assignment.



Since the levels $\ell(c)$ and higher remain poor, there is no vertex of level $\ell(c)$ which c can be assigned to and the bandwidth which is not used at the levels $\ell(c)$ and higher is 0. Then, we calculate the bandwidth which is not used at the levels lower than $\ell(c)$. We call the vertex v a *free-root* if v and its all the children are assignable, and its parent is not assignable. Let T denote the set of free-roots. The non-used bandwidth is the total bandwidth of the vertices of T. Each free-root should be the right child of its parent. For, if a free-root is the left child of its parent p, the right child of p is also assignable because of the compactness. Therefore, p is assignable, which contradicts the definition of free-root. Suppose that there are more than one free-roots f and f' at the same level ℓ, and suppose that f' lies to the right of f. As discussed above, both f and f' are the right children of their parents. Since both f and f' are assignable (and non-dead), all vertices between them are non-dead because of the compactness. In particular, the vertex f'' immediately left of f' is non-dead. Since f' is assignable and f'' is non-dead, the parent p' of f' and f'' has no assigned descendants. Moreover, since f' is assignable, p' has no assigned ancestors. Then p' is assignable, which again contradicts the definition. Hence, in a compact assignment, there is at most one free-root at each level. The total bandwidth is then at most $2^0 + 2^1 + \cdots + 2^{\ell(c)-1}$, which is smaller than $2^{\ell(c)}$. □

As we have mentioned previously, we excluded inputs in which the total bandwidth of all codes which should be assigned exceeds the capacity. Hence, Case (7) does not occur if the semi-compactness is preserved. (Actually, we do not have to exclude this case because our algorithm preserves the semi-compactness and detects this situation, and in such a case, our algorithm can simply reject the insertion.)

The following lemma proves the correctness of EXTENDED-LAZY on insertions.

Lemma 2. EXTENDED-LAZY *preserves the semi-compactness on insertions.*

Proof. In order to prove this lemma, we have to show that the five conditions (i) through (v) of semi-compactness are preserved after an insertion is served. It is relatively easy to show that (i) is preserved because EXTENDED-LAZY uses only AppendRich, AppendPoor, and FreeTail, each of which preserves the orthogonality even by a single application. So, we show that conditions (ii) through (v) are satisfied after the execution of each of Cases (1) through (6), provided that (i) through (v) are satisfied before the execution.

Case (1): It is not hard to see that conditions (iii) through (v) remain satisfied because no tank is created or removed. We check that condition (ii) is satisfied for each group of levels $[0, \ell(c))$, $\ell(c)$, and $(\ell(c), h]$.

- $[0, \ell(c))$: Condition (ii) remains satisfied because nothing changes.
- $\ell(c)$: Since we only append c to the leftmost non-dead vertex at $\ell(c)$, condition (ii) remains satisfied.
- $(\ell(c), h]$: Let v be the vertex to which the code c is assigned. Note that by AppendRich($\ell(c), c$), some of ancestors of v may turn from non-dead to dead. Suppose that vertex v_s of level s ($s > \ell(c)$) turned from non-dead to dead. Then, by the above observation, v_s is an ancestor of v. Next, let v' be the vertex which is immediately left of v. Then, since v' was dead, v_s is not an ancestor of v'. As a result, the ancestor of v' at level s is the vertex, say v'_s, immediately left of v_s, which implies that v'_s was dead. Thus, condition (ii) remains satisfied at any level.

Case (2): We can see that condition (ii) is satisfied by a similar discussion as Case (1). The tank tank$[\ell(c), t]$ is only the tank which is created in Case (2). Then, condition (iii) is satisfied because no level from $\ell(c)$ to t belonged to a tank. Since level $t-1$ was poor and level t was rich, level t contained at least one assinged vertex. Therefore, condition (iv) is satisfied. Also, condition (v) is satisfied because the levels from $\ell(c)$ to $t-1$ were poor.

Case (3): We check that the four conditions (ii) through (v) are satisfied for each group of levels $[0, \ell(c))$, $[\ell(c), b)$, b, (b, s), s, (s, t), t, and $(t, h]$. ($\ell(c)$, b, t, and s are defined in the description of Case (3).)

$[0, \ell(c))$: All the conditions remain satisfied because nothing changes.

$[\ell(c), b)$: All the conditions remain satisfied by the same argument as Case (2).

b: We only append c' and c to the leftmost two non-dead vertices. Hence, condition (ii) is satisfied. Condition (iii) remains satisfied because tank$[b, t]$ is removed and tank$[\ell(c), b]$ is created. Condition (iv) is satisfied because c' is assigned to the vertex immediately right of the vertex to which c is assigned. Also, condition (v) remains satisfied because b is the top of tank$[\ell(c), b]$.

(b, s): The tank tank$[b, t]$ is removed and the statuses of some vertices may turn from non-dead to dead. Since only tank$[b, t]$ is removed, conditions (iii), (iv) and (v) remain satisfied. Also, condition (ii) is satisfied by the same discussion as the case of levels $(\ell(c), h]$ of Case (1).

s: The vertex v to which c'' was assigned remains dead because c' and c are assigned to descendants of v. So, condition (ii) remains satisfied. Next, tank$[b, t]$ is removed while tank$[s, t]$ is created. So, conditions (iii), (iv), and (v) also remain satisfied.

(s, t): The tank tank$[b, t]$ is removed and tank$[s, t]$ is created, but other things do not change. Therefore, all the conditions remain satisfied.

t: The tank tank$[b, t]$ is changed into tank$[s, t]$ if $s \neq t$. In this case, nothing changes and hence all the conditions remain satisfied. If $s = t$, tank$[b, t]$ is removed and code c'' is assigned to the vertex which was tank$[b, t]$. In this case, the statuses of vertices of level t remain the same. Therefore, all the conditions also remain satisfied.

$(t, h]$: All the conditions remain satisfied because nothing changes.

Case (4): By executing FreeTail($\ell(c)$), receiving code c', and executing AppendRich($\ell(c), c$), only tank$[\ell(c'), \ell(c)]$ is removed but the statuses of all vertices remain the same. Hence, all the conditions remain satisfied. Note that at this moment, the situation is the same as the situation just before Case (2) is executed. Then, we can do the same argument as Case (2).

Case (5): Similar to Case (4), but this time, after executing FreeTail($\ell(c)$), receiving code c', and executing AppendRich($\ell(c), c$), the situation is the same as the situation before Case (2) or Case (3) is executed. So, the correctness follows from the same argument as Cases (2) and (3).

Case (6): Similarly as the proof of Case (3), we check conditions (ii) through (v) for each group of levels $[0, b)$, $[b, \ell(c))$, $\ell(c)$, $(\ell(c), s)$, s, (s, t), t, and $(t, h]$. To avoid lengthy description, however, we omit the proof of this case. □

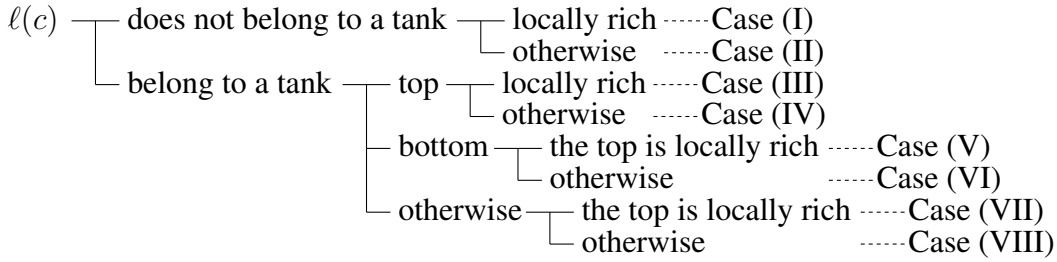
3.2. Executions of EXTENDED-LAZY *for deletions*

Next, we describe executions of EXTENDED-LAZY for deletions. Similarly as Sec. 3.1., for a deletion d_c, there are eight cases depending on the status of level $\ell(c)$ as summarized in Figure 7, each of which

will be explained in the following.

Case (I): The case that $\ell(c)$ does not belong to a tank and is locally rich. Remove c. If c is the rightmost dead vertex at $\ell(c)$, do nothing. Otherwise, use FreeTail($\ell(c)$) and receive a code c' of level $\ell(c)$. Then, using AppendLeft($\ell(c), c'$), assign c' to the vertex to which c was assigned. Note that the vertex v which was the rightmost dead vertex of level $\ell(c)$ becomes non-dead after the above operations, which may turn some dead vertices in the path from v to the root non-dead. As a result, an assignment may become non-semi-compact. If the semi-compactness is broken, we use the operation REPAIR, which will be explained later, to retrieve the semi-compactness. The cost of this case is either 1 or 0, and the number of locally rich levels decreases by one without considering the effect of REPAIR. (We later estimate these quantities considering the effect of REPAIR.)

Figure 7. Execution of EXTENDED-LAZY for a deletion d_c.



Case (II): The case that $\ell(c)$ does not belong to a tank and is not locally rich. EXTENDED-LAZY behaves in exactly the same way as Case (I). Note that vertex v which was the rightmost dead vertex at level $\ell(c)$ becomes non-dead after the above operations, but v is a right child because $\ell(c)$ was not locally rich. Since the semi-compactness was satisfied before the execution, the vertex immediately left of v was (and is) dead, which implies that the parent and hence all ancestors of v are still dead. Thus, we do not need REPAIR in this case. It costs either 1 or 0, and the number of locally rich levels increases by one or remains unchanged because $\ell(c)$ may become locally rich.

Case (III): The case that $\ell(c)$ belongs to tank$[b, t]$, $\ell(c) = t$, and t is locally rich. First, remove c. Next, execute FreeTail(t) and receive the code c' of level b from tank$[b, t]$. If c was assigned to the vertex immediately left of tank$[b, t]$ at t, do nothing. Otherwise, using FreeTail(t), receive a code c'' of level t, and using AppendLeft(t, c''), assign c'' to the vertex to which c was assigned. We then find a level to which we assign the code c'. Starting from level t, we see if the level contains at least one code, until we reach level $b + 1$. Let ℓ be the first such level. Then execute AppendRich(ℓ, c'), which creates tank$[b, \ell]$. If there is no such level ℓ between t and $b + 1$, execute AppendRich(b, c'). In this case, we may need REPAIR. Without considering the effect of REPAIR, it costs either 1 or 2. If it costs 1, the number of locally rich levels stays unchanged or decreases by one, and if it costs 2, the number of locally rich levels decreases by one.

Case (IV): The case that $\ell(c)$ belongs to tank$[b, t]$, $\ell(c) = t$, and t is not locally rich. EXTENDED-LAZY behaves in exactly the same way as Case (III). In this case, we do not need REPAIR by a similar observation as Case (II). It costs either 1 or 2, and the number of locally rich levels increases by one.

Case (V): The case that $\ell(c)$ belongs to tank$[b, t]$, $\ell(c) = b$, and t is locally rich. First, remove c. If c was the code assigned to tank$[b, t]$, stop here; otherwise, do the following: Execute FreeTail(t) and receive the code c' of level b from tank$[b, t]$. Then, using AppendLeft(b, c'), assign c' to the vertex to which c was assigned. In this case, we may need REPAIR because tank$[b, t]$ becomes unassigned. The incurred cost is 1 or 0, and the number of locally rich levels decreases by one without considering the effect of REPAIR.

Case (VI): The case that $\ell(c)$ belongs to tank$[b, t]$, $\ell(c) = b$, and t is not locally rich. EXTENDED-LAZY behaves in exactly the same way as Case (V). In this case, we do not need REPAIR for the same reason as Case (II). The cost is 1 or 0, and the number of locally rich levels increases by one.

Case (VII): The case that $\ell(c)$ belongs to tank$[b, t]$, $b < \ell(c) < t$, and t is locally rich. First, remove c. Next, execute FreeTail(t) and receive the code c' of level b from tank$[b, t]$. If c was assigned to the rightmost assigned vertex at $\ell(c)$, do nothing. Otherwise, using FreeTail($\ell(c)$), receive a code c'' of level $\ell(c)$, and using AppendLeft($\ell(c), c''$), assign c'' to the vertex to which c was assigned. We then find a level to which we assign the request c' in the same way as Case (III). Starting from level $\ell(c)$, we see if the level contains at least one code, until we reach level $b + 1$. Let ℓ be the first such level. Then execute AppendRich(ℓ, c'), which creates tank$[b, \ell]$. If there is no such level ℓ between $\ell(c)$ and $b + 1$, execute AppendRich(b, c'). In this case, we may need REPAIR. Without considering the effect of REPAIR, it costs either 1 or 2. If it costs 1, the number of locally rich levels is unchanged or decreases by one, and if it costs 2, the number of locally rich levels decreases by one.

Case (VIII): The case that $\ell(c)$ belongs to tank$[b, t]$, $b < \ell(c) < t$, and t is not locally rich. EXTENDED-LAZY behaves in exactly the same way as Case (VII). In this case, we do not need REPAIR for the same reason as Case (IV). The cost is 1 or 2. The number of locally rich levels increases by one or two when the cost is 1, and by one when the cost is 2.

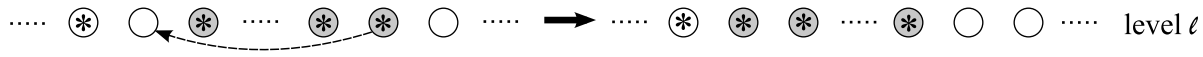
Recall that after executing Cases (I), (III), (V), or (VII), the OVSF code tree may not satisfy semi-compactness. In such a case, however, there is only one level that breaks the conditions of semi-compactness, and furthermore, there is only one broken condition, namely, either (ii) or (v). If (ii) is broken at level ℓ, level ℓ consists of, from left to right, a sequence of unassigned dead vertices up to some point, one non-dead vertex v, a sequence of (at least one) assigned dead vertices, and a sequence of non-dead vertices. Note that, if ℓ is a bottom of a tank tank$[\ell, t]$, the last non-dead vertices include a leftmost level-ℓ descendant of tank$[\ell, t]$ (which is non-dead by definition). This non-dead vertex v was called a "hole" in [6]. We also use the same terminology here, and call level ℓ a *hole-level*. If (v) is broken at level ℓ, ℓ is a bottom of tank$[\ell, t]$ and is rich. Furthermore, level ℓ consists of, from the leftmost vertex, a sequence of one or more unassigned dead vertices, a sequence of one or more non-dead vertices, and then the leftmost level-ℓ descendant of tank$[\ell, t]$. We call level ℓ a *rich-bottom-level*. A level is called a *critical-level* if it is a hole-level or a rich-bottom-level.

The idea of REPAIR is to resolve a critical-level one by one. When we remove a critical-level ℓ by REPAIR, it may create another critical-level. However, we can prove that there arises at most one new critical level, and its level is higher than ℓ. Hence we can obtain a semi-compact assignment by applying REPAIR at most h times.

We explain the operation REPAIR. If ℓ is a hole-level and ℓ is not a bottom of a tank, then we execute FreeTail(ℓ) so that we receive a code c and execute AppendLeft(ℓ, c), i.e., we release the code c assigned

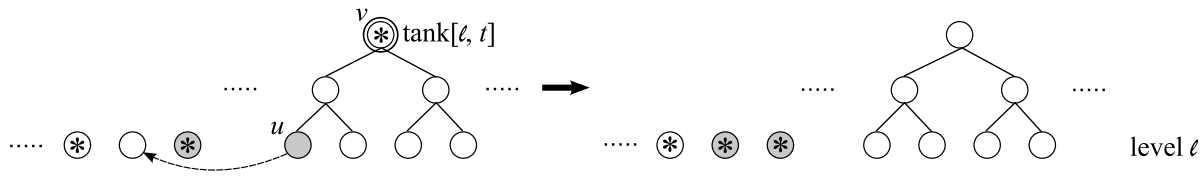
to the rightmost assigned vertex at level ℓ, and reassign c to the hole to fill it. (See Figure 8.) In this case, the cost of REPAIR is 1. If ℓ is locally rich, the number of locally rich levels decreases by one and at most one critical-level may appear, which means that we may need to apply REPAIR once more. Otherwise, the number of locally rich levels increases by one and a critical-level does not appear. If ℓ is a hole-level

Figure 8. REPAIR for a hole-level ℓ which is not a bottom of a tank.



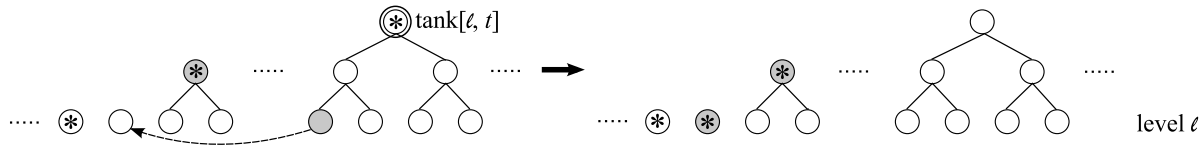
and ℓ is a bottom of a tank v(=tank[ℓ, t]), then there is a vertex u that is the leftmost level-ℓ descendant of v. Recall that the code virtually assigned to v is actually a code for level ℓ and is assigned to u. We release this code c using FreeTail(t) and perform AppendLeft(ℓ, c). (See Figure 9.) In this case, the cost of REPAIR is 1. If t is locally rich, the number of locally rich levels decreases by one and at most one critical-level may appear, which means that we may need to apply REPAIR once more. Otherwise, the number of locally rich levels increases by one and a critical-level does not appear.

Figure 9. REPAIR for a hole-level ℓ which is a bottom of a tank v.



Finally, if ℓ is a rich-bottom-level, then we will do the same operation, namely, release the code c from the tank, and execute AppendLeft(ℓ, c). (See Figure 10.) In this case, the cost of REPAIR is 1. If t is locally rich, the number of locally rich levels decreases by one and at most one critical-level may appear, which means that we may need to apply REPAIR once more. Otherwise, the number of locally rich levels increases by one and a critical-level does not appear.

Figure 10. REPAIR for a rich-bottom-level ℓ.



Lemma 3. EXTENDED-LAZY *preserves the semi-compactness on deletions.*

Proof. Similarly as Lemma 2, we will check that the five conditions (i) through (v) of semi-compactness are preserved for each application of Cases (II), (IV), (VI), and (VIII). For Cases (I), (III), (V), and (VII),

at most one critical-level may appear but the five conditions (i) through (v) must be preserved at all other levels. We also need to verify these facts for REPAIR, but this can be done similarly, and we will omit it here. It is relatively easy to show that condition (i) is preserved because removing codes, and applying AppendRich, AppendPoor, FreeTail, and AppendLeft preserve the orthogonality. So, in the following, we will check conditions (ii) through (v). Similarly as Cases (1) and (3) in the proof of Lemma 2, we check conditions for each group of levels.

Case (I):

$[0, \ell(c))$: Nothing changes. Thus, all the conditions remain satisfied.

$\ell(c)$: Only the rightmost assigned vertex turns unassigned by removing c, and applying FreeTail($\ell(c)$) and AppendLeft($\ell(c), c'$). Therefore, condition (ii) remains satisfied. Since $\ell(c)$ does not belong to any tank, conditions (iii), (iv), and (v) also remain satisfied.

$(\ell(c), h]$: Conditions (iii) and (iv) remain satisfied because no tank is removed or created. In the following, we show that there is at most one level that breaks the conditions (ii) or (v).

Since the rightmost dead vertex v at level $\ell(c)$ turns non-dead and v is the left child of its parent v', v' turns from dead to non-dead. Also, if v' was the rightmost dead vertex at level $\ell(c)+1$ and is the left child of its parent v'', v'' turns from dead to non-dead. Otherwise, v'' remains dead. In this way, we can apply the same argument to the upper levels. Then there is a level ℓ^* such that one vertex turns from dead to non-dead at each level from $\ell(c)$ to ℓ^*, and nothing changes at levels ℓ^*+1 or higher. Clearly, levels ℓ^*+1 or higher do not break the conditions. Also, we can see that both conditions (ii) and (v) are satisfied at any level in $[\ell(c), \ell^*)$ as follows: Consider a level $\ell \in [\ell(c), \ell^*)$ and suppose that v''' at level ℓ turns from dead to non-dead. Condition (ii) is satisfied at ℓ because v''' was the rightmost dead vertex at ℓ. Condition (v) is also satisfied for the following reason: v''' was the rightmost dead vertex at ℓ and is the left child of its parent, so ℓ was locally rich. Since condition (v) was satisfied before the application of Case (I), ℓ must not belong to a tank. So, the conditions can be broken only at ℓ^*. Let $\tilde{v}$ be the vertex at ℓ^* that turns from dead to non-dead. Note that if (ii) is broken, $\tilde{v}$ was not the rightmost dead vertex, while if (v) is broken, $\tilde{v}$ was the rightmost dead vertex (since ℓ^* turns from poor to rich), namely, ℓ^* cannot break both (ii) and (v). This completes the proof.

Case (II):

$[0, \ell(c)]$: All the conditions are satisfied by the same argument as Case (I).

$(\ell(c), h]$: Since v is a right child and its sibling remains dead, the parent of v remains dead. So, the statuses of all vertices remain the same. Thus, all the conditions remain satisfied.

Case (III):

$[0, b)$: Nothing changes and so all the conditions remain satisfied.

$[b, \ell)$: The tank tank$[b, t]$ is removed and tank$[b, \ell]$ is created. However, nothing changes except for the tanks. Thus, all the conditions remain satisfied.

ℓ: If $\ell = t$, tank$[b, t]$ is moved to the vertex immediately left of tank$[b, t]$. Thus, conditions (ii), (iii), and (v) remain satisfied. Also, condition (iv) remains satisfied, because there is at least one assigned vertex except for tank$[b, t]$ at t. If $\ell \neq t$, c' is assigned to the vertex to the right of the rightmost dead vertex which is also assigned and tank$[b, \ell]$ is created. Hence, conditions (ii) and (iv) remain satisfied. Also, condition (iii) remains satisfied because tank$[b, t]$

is removed and tank$[b, \ell]$ is created. Since ℓ is the top of tank$[b, \ell]$, condition (v) also remain satisfied

(ℓ, t): Only tank$[b, t]$ is removed. Therefore, all the conditions remain satisfied.

t: We have already proven above the case of $\ell = t$. Otherwise, tank$[b, t]$ is removed and the rightmost assigned vertex v turns unassigned. Since levels $[\ell, t)$ were poor, there was an assigned ancestor of the leftmost non-dead vertices at levels $[\ell, t)$. The assigned ancestor was v because there were no assigned vertices at levels (ℓ, t). Hence, v remains dead. Condition (ii) remains satisfied because only the rightmost dead vertex turns non-dead. Conditions (iii), (iv), and (v) also remain satisfied because tank$[b, t]$ is removed.

$(t, h]$: We can use the same argument as the proof for levels $(\ell(c), h]$ of Case (I).

Case (IV):

$[0, t]$: All the conditions remain satisfied by the same discussion as Case (III).

$(t, h]$: The parent of tank$[b, t]$ remains dead because tank$[b, t]$ is a right child and its sibling remains dead as we proved in the proof of Case (III). So, the statuses of all vertices remain the same. Thus, all the conditions remain satisfied.

Case (V):

$[0, b)$: Nothing changes and so all the conditions remain satisfied.

$[b, t)$: Only tank$[b, t]$ is removed and all the conditions remain satisfied.

t: The tank tank$[b, t]$ is removed and the rightmost dead vertex, which was tank$[b, t]$, turns non-dead. Condition (ii) remains satisfied because only the rightmost dead vertex turns non-dead. Also conditions (iii), (iv), and (v) remain satisfied because tank$[b, t]$ is removed.

$(t, h]$: We can use the same argument as the proof for level $(\ell(c), h]$ of Case (I).

Case (VI):

$[0, t]$: All the conditions remain satisfied by the same discussion as Case (V).

$(t, h]$: Since tank$[b, t]$ is a right child and its sibling remains dead, the parent of tank$[b, t]$ remains dead. So, the statuses of all vertices remain the same. Thus, all the conditions remain satisfied.

Case (VII):

$[0, b)$: Nothing changes and so all the conditions remain satisfied.

$[b, \ell(c)]$: We can use the same argument as the proof for levels $[b, t]$ of Case (III).

$[\ell(c), t)$: Only tank$[b, t]$ is removed and all the conditions remain satisfied.

t: The tank tank$[b, t]$ is removed and the rightmost dead vertex, which was tank$[b, t]$, turns non-dead. Condition (ii) remains satisfied because only the rightmost dead vertex turns non-dead. Also conditions (iii), (iv), and (v) remain satisfied because tank$[b, t]$ is removed.

$(t, h]$: We can do the same argument as the proof for level $(\ell(c), h]$ of Case (I).

Case (VIII):

$[0, t]$: All the conditions remain satisfied by the same discussion as Case (VII).

$(t, h]$: The statuses of all vertices remain the same by the same discussion as Case (VI).

□

4. Competitive Analyses of EXTENDED-LAZY

First, we estimate the cost and the increase in the number of locally rich levels incurred by applications of REPAIR. By a single application of REPAIR, the cost of 1 is incurred and the number of locally rich levels increases or decreases by one. In case that the number of locally rich levels increases by one, the resulting OVSF code tree is semi-compact. On the other hand, if the number of locally rich levels decreases by one, the resulting OVSF code tree may not be semi-compact and we may need one more application of REPAIR. Hence, if REPAIR is executed k times, then the total cost of k is incurred, and the number of locally rich levels decreases by $k-2$ or k. (In the case of $k=1$, "decreases by $k-2$" means "increases by one".)

Table 1. The costs and increases in the number of locally rich levels for each execution of EXTENDED-LAZY.

Case	(1)	(2)	(3)	(4)	(5)		(6)
Cost	1	1	3	2	2	4	3
Increase	≤ 1	≤ 1	0	≤ 1	≤ 1	0	0

Case	(I)	(II)	(III)		(IV)	(V)	(VI)	(VII)		(VIII)	
Cost	$\leq k+1$	≤ 1	$k+1$	$k+2$	≤ 2	$\leq k+1$	≤ 1	$k+1$	$k+2$	1	2
Increase	$\leq -k+1$	≤ 1	$\leq -k+2$	$\leq -k+1$	1	$\leq -k+1$	1	$\leq -k+2$	$\leq -k+1$	≤ 2	1

Then, we estimate the cost and the increase in the number of locally rich levels for each of the cases (1) through (6) and (I) through (VIII) of EXTENDED-LAZY. From the observations of Sects. 3.1. and 3.2., and the above observation on REPAIR, these quantities can be calculated as in Table 1. There are two values in Case (5): Left and right values correspond to the cases where Cases (2) and (3), respectively, are executed after Case (5). There are also two values in Cases (III), (VII), and (VIII) corresponding to behaviors of EXTENDED-LAZY. In the lower table, k denotes the number of applications of REPAIR. One can see that, from the upper table, the sum of the cost and the increase in the number of locally rich levels is at most 4 for serving an insertion. This occurs when EXTENDED-LAZY executes Case (5) followed by Case (3). Similarly, by the lower table, the sum of the cost and the increase in the number of locally rich levels for serving one deletion is at most 3, which occurs in Cases (III), (IV), (VII), and (VIII).

Now, we are ready to calculate the competitive ratio of EXTENDED-LAZY. For an arbitrary input sequence σ, let I and D be the sets of insertions and deletions in σ, respectively. It is easy to see that the cost of an optimal offline algorithm is at least $|I|$ because each insertion incurs a cost of 1 in any algorithm. We then estimate the cost of EXTENDED-LAZY. For $i \in I$ and $d \in D$, let e_i and e_d be the costs of EXTENDED-LAZY for serving i and d, respectively. The cost of EXTENDED-LAZY for σ is then $\sum_{i\in I} e_i + \sum_{d\in D} e_d$. Also, for $i \in I$ and $d \in D$, let q_i and q_d be the increases in the number of locally rich levels caused by EXTENDED-LAZY in serving i and d, respectively. Define Q to be the number of locally rich levels in the OVSF code tree at the end of the sequence σ. Then, $Q = \sum_{i\in I} q_i + \sum_{d\in D} q_d$

since there is no locally rich level at the beginning. The cost of EXTENDED-LAZY for σ is

$$\begin{aligned}
\sum_{i\in I} e_i + \sum_{d\in D} e_d &\leq \sum_{i\in I} e_i + \sum_{d\in D} e_d + Q \\
&= \sum_{i\in I}(e_i + q_i) + \sum_{d\in D}(e_d + q_d) \\
&\leq \sum_{i\in I} 4 + \sum_{d\in D} 3 \qquad (1) \\
&= 4|I| + 3|D| \\
&\leq 7|I|. \qquad (2)
\end{aligned}$$

(1) is due to the above analysis, and (2) is due to the fact that $|D| \leq |I|$ since for each deletion, there must be a preceding insertion corresponding to it. Now, the following theorem is immediate from the above inequality.

Theorem 1. *The competitive ratio of* EXTENDED-LAZY *is at most 7.*

Next, we give a lower bound on the competitive ratio of EXTENDED-LAZY.

Theorem 2. *The competitive ratio of* EXTENDED-LAZY *is at least* $7 - \varepsilon$ *for any positive constant* ε.

Proof. Consider an OVSF code tree of height h ($h \geq 5$). The number of leaves n is 2^h. First, we give following requests in this order (which we call the *initialization sequence*): $n/4$ insertions of level-0 codes, an insertion of a level-1 code, an insertion of a level-0 code, an insertion of a level-i code for $i = 2, 3, \ldots, h-2$ ($h-3$ insertions in total), and an insertion of a level-2 code. Note that after the initialization sequence, two tanks (tank[0, 1] and tank[2, $h-2$]) are created at the $(n/4+2)$-th and the final request, respectively. EXTENDED-LAZY can serve each insertion with cost 1, thus the cost of EXTENDED-LAZY is $n/4 + h$. Note that the cost of an optimal offline algorithm OPT is also $n/4 + h$.

Next, we consider the *I-sequence*, which is defined as the sequence of $h-4$ insertions of level-i codes for $i = 1, \ldots, h-4$ (in the increasing order of i). When the I-sequence is given to EXTENDED-LAZY after the initialization sequence, EXTENDED-LAZY executes Case (5) followed by Case (3) for every insertion in the I-sequence. We then define the *D-sequence*. It consists of the $h-4$ deletions of level-i codes for $i = h-4, \ldots, 1$ (in the decreasing order of i). Here, each deletion requires to remove the code which is assigned to the leftmost vertex among all assigned vertices in the corresponding level. If the D-sequence is given to EXTENDED-LAZY subsequently to the I-sequence, then EXTENDED-LAZY executes Case (VIII) for each deletion in the D-sequence. Finally, we define the *T-sequence* which consists of two requests; the former is a deletion of a level-2 code, which requires to remove the code assigned to the leftmost assigned vertex of level 2, and the latter is an insertion of a level-2 code. For each of the T-sequence, given after the D-sequence, EXTENDED-LAZY executes Cases (I) and (2), respectively. It should be noted that, for EXTENDED-LAZY, the configuration of the OVSF code tree at this moment is the same as the one immediately after processing the initialization sequence.

The complete sequence is as follows: We first give the initialization sequence. Then, we repeat k times the concatenation of I-sequence, D-sequence, and T-sequence. Careful calculation shows that the cost of EXTENDED-LAZY for each concatenation is $k(7h - 26)$, while the cost of an optimal offline

algorithm OPT is $k(h-3)$. Hence, the ratio between the costs of EXTENDED-LAZY and OPT is $\frac{n/4+h+k(7h-26)}{n/4+h+k(h-3)} = 7 - \frac{3n/2+6h+5k}{n/4+h+k(h-3)}$. Note that $3n/2+6h+5k$ and $n/4+h+k(h-3)$ are positive because $h \geq 5$ and $k > 0$. Now, if h and k go infinity, $\frac{3n/2+6h+5k}{n/4+h+k(h-3)}$ becomes smaller than any positive constant ε. □

5. A Lower Bound

Theorem 3. *For any positive constant ε, there is no $(2-\varepsilon)$-competitive online algorithm for the online OVSF code assignment problem.*

Proof. Consider an OVSF code tree of height h (where h is even), namely, the number of leaves are $n = 2^h$. First, an adversary gives n insertions of level-0 codes so that the vertices of level 0 are fully assigned, by which, an arbitrary online algorithm incurs the cost of n. Then, depending on the assignment of the online algorithm, the adversary requires to remove one code from each subtree rooted at a vertex of level $h/2$. (Hereafter, we simply say "subtree" to mean a subtree of this size.) Since there are $\sqrt{n}$ such subtrees, the adversary gives $\sqrt{n}$ deletions in total. Next, the adversary gives an insertion i_{c_1} of a level-$h/2$ code. To assign c_1, the online algorithm has to make one of subtrees empty by reassignments, for which the cost of at least $\sqrt{n}-1$ is required.

Again, depending on the behavior of the online algorithm, the adversary requires to remove $\sqrt{n}$ codes of level 0 *uniformly* from each subtree except for the subtree to which c_1 is assigned. Here, "uniformly" means that the numbers of removed codes for any pair of subtrees differ by at most 1; in the current case, the adversary requires to remove two codes from one subtree, and one code from each of the other $\sqrt{n}-2$ subtrees. Subsequently, the adversary gives an insertion i_{c_2} of a level-$h/2$ code. Similarly as above, the online algorithm requires at least $\sqrt{n}-2$ reassignments to assign c_2.

The adversary repeats the same operation $\sqrt{n}$ rounds, where one round consists of $\sqrt{n}$ deletions to remove codes of level 0 uniformly from subtrees, and one insertion of a level-$h/2$ code. Eventually, all initial codes of level 0 are removed, and the final OVSF code tree contains $\sqrt{n}$ codes of level $h/2$.

The total cost of the online algorithm is at least

$$
\begin{aligned}
n+\sqrt{n}+\sum_{i=1}^{\sqrt{n}}\left(\sqrt{n}-\left\lceil\frac{\sqrt{n}}{\sqrt{n}+1-i}\right\rceil\right) &= n+\sqrt{n}+\sum_{i=1}^{\sqrt{n}}\left(\sqrt{n}-\left\lceil\frac{\sqrt{n}}{i}\right\rceil\right)\\
&= 2n+\sqrt{n}-\sum_{i=1}^{\sqrt{n}}\left\lceil\frac{\sqrt{n}}{i}\right\rceil\\
&> 2n+\sqrt{n}-\sum_{i=1}^{\sqrt{n}}\left(\frac{\sqrt{n}}{i}+1\right)\\
&= 2n-\sum_{i=1}^{\sqrt{n}}\frac{\sqrt{n}}{i}\\
&= 2n-\sqrt{n}\left(\log\sqrt{n}+\left(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}\right)\right).
\end{aligned}
$$

On the other hand, the cost of an optimal offline algorithm is $n+\sqrt{n}$ since it does not need reassignment. Hence, the competitive ratio is at least

$$\frac{2n-\sqrt{n}(\log\sqrt{n}+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}} = 2-\frac{\sqrt{n}(\log\sqrt{n}+2+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}}.$$

Since $\lim_{n\to\infty}(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}) = \gamma$ ($\gamma \simeq 0.577$) is the Euler's constant, the term $\frac{\sqrt{n}(\log\sqrt{n}+2+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}}$ becomes smaller than any positive constant ε as n goes infinity. □

6. Conclusion

We proposed a new online algorithm for the OVSF code assignment problem. We proved that it is 7-competitive, improving the previous bound of 10, and we showed that our analysis is tight by giving an input sequence for which the competitive ratio of our algorithm is $7-\varepsilon$ for an arbitrary constant $\varepsilon > 0$. We also improved the lower bound from 5/3 to 2. The upper bound is now improved to 6 by Chin, Ting, and Zhang [12]. However, the gap between upper and lower bounds is still large and it is an interesting future work to narrow it.

References

1. Holma, H.; Toskala, A. *WCDMA for UMTS*; Wiley: New York, NY, USA, 2001.
2. Laiho, J.; Wacker, A.; Novosad, T. *Radio Network Planning and Optimisation for UMTS*; Wiley: New York, NY, USA, 2002.
3. Adachi, F.; Sawahashi, M.; Okawa, K. Tree-structured generation of orthogonal spreading codes with different lengths for forward link of DS-CDMA mobile radio. *Electronics Letters* **1997**, *33*, 27–28.
4. Erlebach, T.; Jacob, R.; Mihaľák, M.; Nunkesser, M.; Szabó, G.; Widmayer, P. An algorithmic view on OVSF code assignment. *Algorithmica* **2007**, *47*, 269–298.
5. Forišek, M.; Katreniak, B.; Katreniaková, J.; Královič, R.; Koutný, V.; Pardubská, D.; Plachetka, T.; Rovan, B. Online bandwidth allocation. *Proc. of The 16th Annual European Symposium on Algorithms (ESA)* **2007**, *LNCS 4698*, 546–557.
6. Chin, F.Y.L.; Ting, H.F.; Zhang, Y. A constant-competitive algorithm for online OVSF code assignment. *Proc. of The 18th International Symposium on Algorithms and Computation (ISAAC)* **2007**, *LNCS 4835*, 452–463.
7. Chin, F.Y.L.; Zhang, Y.; Zhu, H. Online OVSF code assignment with resource augmentation. *Proc. of The 3rd International Conference on Algorithmic Aspects in Information and Management (AAIM)* **2007**, *LNCS 4508*, 191–200.
8. Chan, J.W.; Chin, F.Y.L.; Ting, H.F.; Zhang, Y. Online Tree Node Assignment with Resource Augmentation. *Proc. of The 15th International Computing and Combinatorics Conference (COCOON)* **2009**, *LNCS 5609*, 358–367.
9. Karakoc, M.; Kavak, A. Stochastic Methods for Dynamic OVSF Code Assignment in 3G Networks *Proc. of The 4th International Symposium on Stochastic Algorithms: Foundations and Applications (SAGA)* **2007**, *LNCS 4665*, 142–153.

10. Minn, T.; Siu, K.Y. Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. *IEEE Journal on Selected Areas in Communications* **2000**, *18*, 1429–1440.
11. Erlebach, T.; Jacob, R.; Tomamichel, M. Algorithmische Aspekte von OVSF code assignment mit Schwerpunkt auf offline code assignment. *Student thesis as ETH Zürich*.
12. Chin, F.Y.L.; Ting, H.F.; Zhang, Y. Constant-Competitive Tree Node Assignment. manuscript.